Recursion and Structural Induction

Mingxun Zhou (mingxunz@ust.hk)

COMP2711 Discrete Mathematical Tools

Learning objectives

- Give recursive definitions with enough base values and rules on smaller inputs.
- Prove properties of recursively built expressions and trees by structural induction.
- Model and solve simple recurrences by expansion, and verify formulas or bounds by induction.

1 Recursive definitions

Recursion defines an object using smaller objects; induction proves properties by following the same structure. We use $\mathbb{N} = \{0, 1, 2, \ldots\}$.

MUST UNDERSTAND Recursive definition

A recursive definition gives **base cases** and **rules for building new values or objects from smaller ones**. A sequence needs enough base values to determine every term.

1.1 Example: a recursive sequence

The recursive definition

$$a_0 = 3, \qquad a_n = 2a_{n-1} + 3 \quad (n \geq 1)$$

gives $3, 9, 21, 45, 93, \ldots$: double the previous term and add 3. We will derive its closed form in Section 3.

1.2 Example: climbing stairs

Let C_n count climbs of n stairs using steps of size 1 or 2. The bases are $C_0 = C_1 = 1$, counting the empty climb at $n = 0$. For $n \geq 2$, split by the last step:

$$\underbrace{C_{n-1}}_{\text{append a 1-step}} + \underbrace{C_{n-2}}_{\text{append a 2-step}} = C_n.$$

The two groups are disjoint and cover every climb. This Fibonacci recurrence gives

n	0	1	2	3	4	5	6
C_n	1	1	2	3	5	8	13

For $n = 4$: $1111, 112, 121, 211, 22$.

2 Structural induction

2.1 Recursively defined arithmetic expressions

The expression $(x + y) \times z$ is valid, while $(x+) \times z$ is invalid. We can describe all valid expressions using recursive rules:

1. Every variable $x, y, z, \ldots$ is a valid expression.
2. If E is valid, then (E) is valid.
3. If E and F are valid, then $E + F$, $E - F$, $E \times F$, and $E \div F$ are valid.

Only finite constructions using these rules are included. For example, build $x + y$ from x, y, wrap it as $(x + y)$, then combine it with z to obtain $(x + y) \times z$.

MUST UNDERSTAND Structural induction

To prove a property of every recursively generated object:

1. prove it for every base object;
2. for each construction rule, assume it for the input objects and prove it for the constructed object.

The method follows by induction on the number of construction steps.

2.2 Example: counting parentheses

Every valid arithmetic expression has equally many opening and closing parentheses. Let $L(E)$ and $R(E)$ be these two counts in expression E.

Base. A variable has no parentheses, so $L(x) = R(x) = 0$.

Parentheses rule. Assume $L(E) = R(E)$. Wrapping adds one opening and one closing parenthesis:

$$L((E)) = L(E) + 1 = R(E) + 1 = R((E)).$$

Operation rules. Assume $L(E) = R(E)$ and $L(F) = R(F)$. For any $\circ \in \{+, -, \times, \div\}$,

$$L(E \circ F) = L(E) + L(F) = R(E) + R(F) = R(E \circ F).$$

Checking validity recursively. Ignore spaces and undo the last construction step:

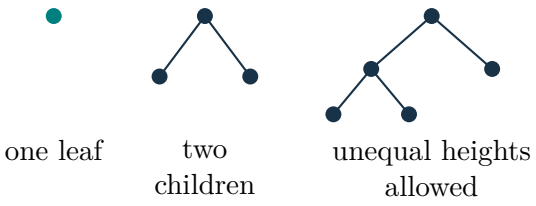
Valid(s)
  if s is a single variable: **return true**
  if $s = (E)$ with one matching outer pair:
    return Valid(E)
  for each split $s = E \circ F$, $\circ \in \{+, -, \times, \div\}$:
    if Valid(E) **and** Valid(F): **return true**
  return false

The outer pair must enclose the entire input; otherwise try each operator as a split. Each recursive call uses fewer symbols, so the procedure terminates. It accepts exactly the generated expressions: a successful call reverses a construction rule, and every valid expression has such a construction. The empty input returns false.

2.3 Example: full binary trees

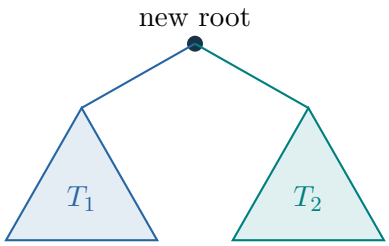
A rooted tree has a distinguished root. A vertex's immediate descendants are its children; a leaf has none. In a *full binary tree*, every non-leaf has exactly two children. Recursively:

1. A single leaf is a full binary tree.
2. Attach two full binary trees T_1, T_2 to a new root to obtain another full binary tree.
3. No other trees are included.



The subtrees need not have equal sizes or heights. Let $v(T)$ be the number of vertices, and let $h(T)$ be the largest number of edges on a root-to-leaf path. For a leaf, $v(T) = 1, h(T) = 0$. Otherwise,

$$v(T) = 1 + v(T_1) + v(T_2), \qquad h(T) = 1 + \max\{h(T_1), h(T_2)\}.$$



Theorem. Every full binary tree satisfies $v(T) \le 2^{h(T)+1} - 1$.

Base: a leaf satisfies $1 = 2^{0+1} - 1$.

Step: assume the bound for T_1, T_2. Put $h = h(T)$. Both subtrees have height at most $h - 1$, so

$$\begin{aligned} v(T) &= 1 + v(T_1) + v(T_2) \\ &\le 1 + (2^{h(T_1)+1} - 1) + (2^{h(T_2)+1} - 1) \\ &\le 1 + (2^h - 1) + (2^h - 1) = 2^{h+1} - 1. \end{aligned}$$

2.4 Further practice: balanced parentheses

Define a set B of strings using the base $\varepsilon \in B$ and the rules

$$u, v \in B \Rightarrow uv \in B, \qquad u \in B \Rightarrow (u) \in B,$$

with no other members. For example, construct (), then ()(), then (()()). Structural induction shows equal numbers of opening and closing parentheses: the base has zero of each, concatenation adds the counts, and wrapping adds one of each. Equal counts alone do not imply membership:)(is not generated. As further practice, characterize membership using the counts in every prefix.

3 Solving recurrences

MUST UNDERSTAND Recurrence relations

A *recurrence relation* expresses a term using earlier terms. A *closed form* expresses it directly using its index. For example, $s_0 = 0$, $s_n = s_{n-1} + 2n - 1$ has closed form $s_n = n^2$.

Solving by expansion. Substitute repeatedly, stop at the base, and sum the terms. Induction can verify the resulting formula.

3.1 Example: expanding a recurrence

Return to $a_0 = 3$, $a_n = 2a_{n-1} + 3$. After j substitutions,

$$a_n = 2^j a_{n-j} + 3 \sum_{i=0}^{j-1} 2^i.$$

Set $j = n$ to reach a_0:

$$a_n = 3 \times 2^n + 3(2^n - 1) = 6 \times 2^n - 3.$$

For verification, the base gives $6 - 3 = 3$. If $a_k = 6 \times 2^k - 3$, then

$$a_{k+1} = 2(6 \times 2^k - 3) + 3 = 6 \times 2^{k+1} - 3.$$

More generally, if $a_0 = A$ and $a_n = ba_{n-1} + c$, then

$$a_n = Ab^n + c\frac{b^n - 1}{b - 1} \quad (b \ne 1), \qquad a_n = A + cn \quad (b = 1).$$

3.2 Example: recursive binary search

For sorted A, search the inclusive range $[\texttt{low}, \texttt{high}]$. Return an index of `target`, or -1 if absent. Start with BinarySearch$(A, \texttt{target}, 1, |A|)$.

BinarySearch$(A, \texttt{target}, \texttt{low}, \texttt{high})$

1: **if** $\texttt{low} > \texttt{high}$ **return** -1 ▷ empty range
2: $\texttt{mid} \leftarrow \lfloor (\texttt{low} + \texttt{high})/2 \rfloor$
3: **if** $A[\texttt{mid}] = \texttt{target}$ **return** mid
4: **if** $\texttt{target} < A[\texttt{mid}]$ **then**
5: **return** BinarySearch$(A, \texttt{target}, \texttt{low}, \texttt{mid} - 1)$
6: **return** BinarySearch$(A, \texttt{target}, \texttt{mid} + 1, \texttt{high})$

Sorted order lets us discard the half that cannot contain the target. The range shrinks at each call, ensuring termination. Passing indices avoids copying: each call does constant extra work.

For powers of two, consider $T_{\text{binary}}(1) = 1$ and $T_{\text{binary}}(n) = T_{\text{binary}}(n/2) + 1$. This models a search that keeps one half and pays one unit per level. After j expansions, $T_{\text{binary}}(n) = T_{\text{binary}}(n/2^j) + j$. Stop at $j = \log_2 n$ to obtain $T_{\text{binary}}(n) = 1 + \log_2 n$.

Thus the worst-case running time is $\Theta(\log n)$. The exact formula above belongs to the simplified model.

3.3 Example: recursive merge sort

Return a sorted copy of A. The linear-time procedure Merge repeatedly takes the smaller front element of two sorted lists, then appends the remaining elements.

MergeSort(A)

1: $n \leftarrow |A|$
2: **if** $n \leq 1$ **return** A ▷ already sorted
3: $\texttt{mid} \leftarrow \lfloor n/2 \rfloor$
4: $\texttt{sortedLeft} \leftarrow$ MergeSort$(A[1 \ldots \texttt{mid}])$
5: $\texttt{sortedRight} \leftarrow$ MergeSort$(A[\texttt{mid} + 1 \ldots n])$
6: **return** Merge$(\texttt{sortedLeft}, \texttt{sortedRight})$ ▷ combine

Both recursive inputs are smaller. The base is sorted, and merging sorted recursive results preserves sorted order. Slices are copies, so splitting and merging together take $\Theta(n)$ work.

For powers of two, the running time satisfies

$$T_{\text{merge}}(1) = \Theta(1), \qquad T_{\text{merge}}(n) = 2T_{\text{merge}}(n/2) + \Theta(n).$$

At internal level j, there are 2^j subproblems of size $n/2^j$. Their total work is $2^j \Theta(n/2^j) = \Theta(n)$, with the same positive bounding constants at every level. There are $\log_2 n$ internal levels and n leaves, each costing $\Theta(1)$. Therefore

$$T_{\text{merge}}(n) = \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

3.4 The Master Theorem

Binary search does constant work per level; merge sort does linear work per level. The Master Theorem also handles recurrences whose level costs shrink or grow.

Theorem (Master Theorem: polynomial work)**.** Fix integers $a \geq 1$, $b \geq 2$ and real constants $c > 0$, $d \geq 0$. For $n = b^k$, let $T(1) > 0$ be a fixed constant and

$$T(n) = aT(n/b) + cn^d \qquad (k \geq 1).$$

Then, as n tends to infinity through powers of b,

$$T(n) = \begin{cases} \Theta(n^d), & a < b^d, \\ \Theta(n^d \log n), & a = b^d, \\ \Theta(n^{\log_b a}), & a > b^d. \end{cases}$$

Here a counts calls, each input shrinks by a factor b, and cn^d is the work outside the calls. This polynomial-work version follows Rosen Section 8.3 [1], with tight bounds on powers of b.

Proof. Put $k = \log_b n$ and $q = a/b^d$. At internal level j, there are a^j calls of size n/b^j, so their total nonrecursive work is

$$a^j c(n/b^j)^d = cn^d q^j.$$

At level k, there are $a^k = n^{\log_b a}$ leaves, each costing $T(1)$. Consequently,

$$T(n) = T(1)a^k + cn^d \sum_{j=0}^{k-1} q^j.$$

- If $q < 1$, then $1 \leq \sum_{j=0}^{k-1} q^j \leq 1/(1-q)$. Also $a^k \leq b^{dk} = n^d$. Thus the root already contributes cn^d, and the entire cost is $\Theta(n^d)$.
- If $q = 1$, each internal level costs cn^d. The sum is k, and $a^k = n^d$, giving $T(n) = n^d(T(1) + ck) = \Theta(n^d \log n)$.
- If $q > 1$, the geometric sum is $\Theta(q^k)$. Since $n^d q^k = a^k$, the internal work and the leaf work are both $\Theta(a^k) = \Theta(n^{\log_b a})$.

For binary search, $(a, b, d) = (1, 2, 0)$, so $a = b^d$ and $T(n) = \Theta(\log n)$. For merge sort, $(a, b, d) = (2, 2, 1)$, again $a = b^d$, giving $T(n) = \Theta(n \log n)$.

The bounds also hold for $\Theta(n^d)$ work outside the calls: use constant upper and lower bounds in the same sum. If needed, absorb finitely many small inputs into positive constant base costs.

4 Practice

1. Define $q_n = 2^n - 1$ recursively for $n \geq 0$.
2. Calculate C_5 and explain why $C_0 = 1$ is appropriate in the stair model.
3. Prove that every full binary tree has one more leaf than internal vertices.
4. Define the positive multiples of 3 recursively by starting with 3 and closing under addition. Prove that the generated set is exactly the set of positive multiples of 3.
5. Let $S(1) = 2$ and $S(n) = 3S(n/3) + n$ for powers of three. Find and verify a closed form.
6. For powers of three, solve $R(1) = 1$, $R(n) = 9R(n/3) + n^2$.

Answers

1. $q_0 = 0$ and $q_n = 2q_{n-1} + 1$ for $n \geq 1$, because $2(2^{n-1} - 1) + 1 = 2^n - 1$.
2. $C_5 = C_4 + C_3 = 5 + 3 = 8$. The empty sequence is the unique climb of zero stairs. In particular, appending one 2-step to it counts the climb 2 when $n = 2$.
3. Let L, I count leaves and internal vertices. The base gives $L = 1, I = 0$. For a new root joining T_1, T_2, the hypothesis gives

$$L = L_1 + L_2 = (I_1 + 1) + (I_2 + 1) = (1 + I_1 + I_2) + 1 = I + 1.$$

4. Structural induction proves every generated number is a positive multiple of 3: this holds for 3 and is preserved by addition. Conversely, ordinary induction on $k \geq 1$ generates $3k$: start with 3, then add 3 to a generated $3k$ to obtain $3(k + 1)$. Both inclusions are needed.
5. After j expansions, $S(n) = 3^j S(n/3^j) + jn$. With $j = \log_3 n$, this gives $S(n) = 2n + n \log_3 n$. At $n = 1$ it is 2; substitution yields $3[2n/3 + (n/3) \log_3(n/3)] + n = 2n + n \log_3 n$.
6. Each internal level costs n^2. There are $\log_3 n$ levels and $9^{\log_3 n} = n^2$ leaves, each of cost 1. Thus $R(n) = n^2(1 + \log_3 n) = \Theta(n^2 \log n)$. The base is 1; substituting at $n/3$ gives $n^2[1 + \log_3 n - 1] + n^2 = n^2(1 + \log_3 n)$.

5 Additional reading

Hanoi. Move the top $n-1$ disks to the spare peg, move the largest disk to the target, then move the $n-1$ onto it. The move count satisfies $H(0)=0$ and $H(n)=2H(n-1)+1$, hence $H(n)=2^n-1$ by expansion or induction. The recursive calls move only smaller disks; the largest-disk move is legal because its target peg is empty. This also gives an inductive correctness proof.

Fibonacci computation. Use $F_0=0, F_1=1$. Naive recursion recomputes earlier values: its call count satisfies $D_0=D_1=1$ and $D_n=D_{n-1}+D_{n-2}+1$, with solution $D_n=2F_{n+1}-1$. Bottom-up computation instead uses $O(n)$ additions, storing two previous values. This counts arithmetic operations; the integers grow with n.

Euclid and Fibonacci. For $t \geq 2$ divisions, reverse the remainder chain: its last two positive terms are at least $1, 2$, and earlier terms dominate the sum of the next two. Thus $b \geq F_{t+1}$, giving logarithmic division count for $b \geq 2$. Handle $t=1$ separately. Rosen's Lamé theorem gives a precise digit-based bound.

Generalized induction. Induction also works on other well-founded orders, such as ordering pairs $(m,n) \in \mathbb{N}^2$ first by n, then by m.

5.1 Strings and concatenation

For an alphabet Σ, let Σ^* contain the empty string ε and every string obtained by repeatedly appending a symbol $x \in \Sigma$. For example, $\varepsilon \to 1 \to 10 \to 101$ when $\Sigma=\{0,1\}$. Define length and concatenation recursively:

$$|\varepsilon|=0, \quad |wx|=|w|+1, \qquad u\varepsilon=u, \quad u(wx)=(uw)x.$$

Claim: $|uv|=|u|+|v|$ for all strings u, v. Fix u and induct structurally on v. The base $v=\varepsilon$ gives $|u\varepsilon|=|u|=|u|+0$. For the step, assume $|uw|=|u|+|w|$. Then

$$|u(wx)|=|(uw)x|=|uw|+1=|u|+|w|+1=|u|+|wx|.$$

See Rosen 5.3 [1] for the core material, 5.4 for recursive algorithms, and 8.1–8.3 for further recurrence methods.

References

[1] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill Education, eighth edition, 2019.